

COM S 352 (Fall 2019)
Project 2
Concurrent Red-Black Trees
Due on 11:59 pm, Nov 18, 2019


Context (red-black trees)

A *red-black tree* is a kind of self-balancing *binary search tree*. Each node in a red-black tree has four attributes:

1. The *key*;
2. Pointer to the right child;
3. Pointer to the left child, and
4. The *color bit* that is interpreted as the color (red or black) of the node.

The color bits are used to ensure the tree remains approximately balanced during insertions and deletions. Balance is preserved by painting each node of the tree with one of two colors in a way that satisfies certain properties, which collectively constrain how unbalanced the tree can become in the worst case. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently.

More specifically, in addition to the requirements imposed on a binary search tree, the following must be satisfied by a red-black tree:

- Each node is either red or black.
- The root is black.
- All leaves (NIL ) are black.
- If a node is red, then both its children are black.
- Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes.

For example, Figure 1 shows a sample red-black tree.

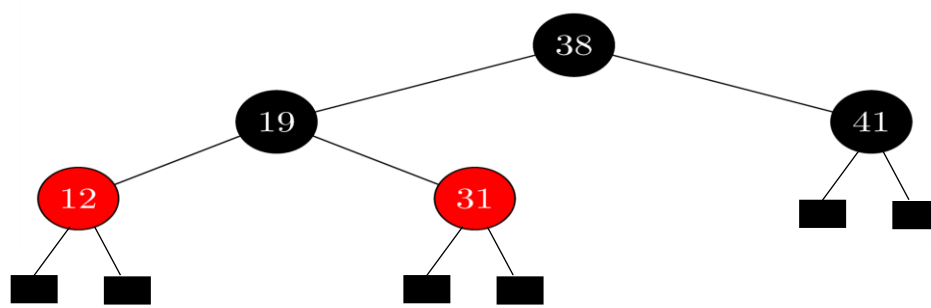


Figure 1. A sample red-black tree

Operations on red-black trees are the following:

- **Search.** Searching for a key is done in the same way as in a standard binary search tree.
- **Insertion.** Inserting a node should preserve all properties of the tree. For example, inserting **key 8** in the tree in Figure 1 will result in the tree in Figure 2.

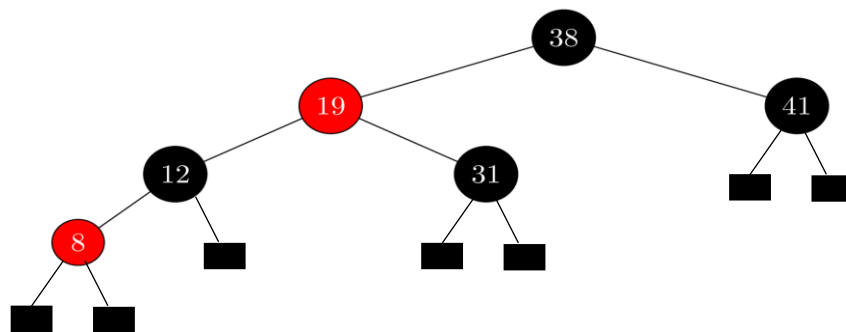


Figure 2: After Inserting 8

- **Deletion.** Deleting a node should preserve all properties of the tree. For example, Deleting **key 8** in the tree in Figure 2 will result in the tree in Figure 3.

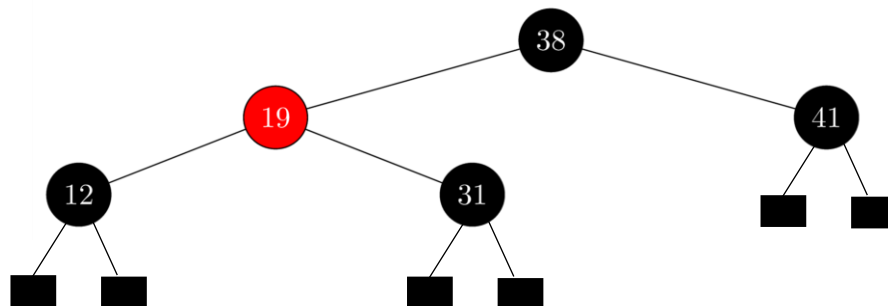


Figure 3: After Deleting 8

Assignment

In this project, you are to develop a solution to the operations of red-black trees that allows multiple concurrent threads to safely operate on a *shared* red-black tree. Each thread should be able to invoke search, insert, and delete operations. A *course-grained* way to implement a concurrent data structure such as a red-black tree is to lock the entire tree in the beginning of an operation and unlock it when the operation ends. For example:

```
void delete(int key) {
    wait(sem);
    // do the operation
    signal(sem);
}
```

The main problem with such a solution is efficiency. This solution simply enforces atomic execution of all the operations, which results in no concurrency.

Your solution should have the following characteristics:

- Your code will be written in C++ and by using the pthreads library.
- It will have *fine-grained* concurrency, meaning that the operations will only enforce mutual exclusion on the elements needed to complete the operation. To this end, in addition to the four attributes mentioned above, you will add a semaphore (or mutex) attribute that enforces mutual execution on accessing that element:

```
struct node {
    int key;
    struct node* left;
    struct node* right;
    bool color;
    mutex m;
}
```

For example, for deleting a node, one can lock the node and its parent, while updating the pointers.

- Similar to the readers/writers problem:
 - Multiple calls to search *only* should not enforce mutual exclusions, i.e., multiple search operations can run concurrently without synchronization.
 - However, concurrent invocation of search and insert/delete should be synchronized.
 - Search should have priority over insertion and deletion.

Input/Output

Your program should take as input a file that contains the following: (1) the initial content of the tree (it can be null) in prefix order, (2) a sequence of thread creations, and (3) a sequence of invocation to the tree operations. This file format should be the same as the following example:

```
3b, 7b, 8r, 10b, 11r, 18r, 22b, 26r

thread1
thread2
thread3

thread1, search(10) || thread2, delete(10) || thread3, insert(15) ||
thread1, insert(5)    || thread3, search(20)
```

All the threads should be created in the main thread. They have to wait until all threads are created. Each thread reads the third part of the file (i.e., operation invocations) and executes its associated invocations in total order. In the above example, thread1 executes `search(10)` and `insert(5)` in order while thread2 executes `insert(15)` and `search(20)` in order.

The output of your program should include the following in an output file:

- Execution time
- The output of each search operation (e.g., `thread1, search(10) -> true`)
- The final red-black tree.

Observe that due to nondeterminism, there may not be a unique solution to a given input. For example, if `thread2, delete(10)` executes before `thread1, search(10)`, then result of search will be false, otherwise, true.

Rules

- If your program doesn't compile or if it uses course-grained synchronization, you will not receive any credit.
- We will provide 5 test cases with different sized trees. Your program is expected to finish execution within the specified time. Otherwise, you will lose 50% of the total credit.

Submission

- You should use C++ to develop the code.
- You should work on this project **individually**.
- You need to turn in electronically by submitting a zip file named: Firstname_Lastname_Project2.zip.
- **Source code must include proper documentation to receive full credit (you will lose 10% of your score, if the code is not well documented).**
- All projects require the use of a **make file** or a certain script file (accompanying with a **readme file** to specify how to use the script/make file to compile), such that the grader will be able to compile/build your executable by simply typing “make” or some simple command that you specify in your readme file.
- **Source code must compile and run correctly on the department machine "pyrite", which will be used by the TA for grading. If your program compiles, but does not run correctly on pyrite, you will lose 15% of your score. If your program doesn't compile at all on pyrite, you will lose 50% of your score (only for this issue/error, points will be deducted separately for other errors, if any).**
- You are responsible for thoroughly testing and debugging your code. The TA may try to break your code by subjecting it to bizarre test cases.
- You can have multiple submissions, but the TA will grade only the last one.

Start as early as possible!